Web Services Made Easier

The Java™ APIs for XML

A Technical White Paper

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

June 2001 Revision 1

Send comments about this document to: xml-feedback@sun.com

Please
Recycle

Introduction and Overview

XML makes data portable. The Java™ platform makes code portable. The Java APIs for XML make it easy to use XML. Put these together, and you have the perfect combination: portability of data, portability of code, and ease of use. In fact, with the Java APIs for XML, you can get the benefits of XML with little or no direct use of XML.

Enterprises are rapidly discovering the benefits of using XML for the integration of data, both internally for sharing legacy data among departments, and externally for sharing data with other enterprises. And because of the data integration that XML offers, it has become the underpinning for Web-related computing.

The really hard part of developing web services is programming the infrastructure, or "plumbing," such as security, messaging capabilities, distributed transaction management, and connection pool management. Another difficulty is that web services must be able to handle huge numbers of users simultaneously, so applications must be highly scalable. These requirements are exactly what the Java™ 2 platform, Enterprise Editon (J2EE™) offers. Add to this the fact that the J2EE platform is a proven technology with multiple vendors offering compatible products today, and it is a "no-brainer" that the J2EE platform is the best platform for deploying web services. And with the new Java APIs for XML, developing web services is getting easier and easier.

The goal of this paper is to make clear what the Java APIs for XML do and how they make writing web applications easier. The paper describes each of the APIs individually and then presents a scenario that shows how they work together. It also mentions various other technologies currently available and how they can be used in conjuction with the Java APIs for XML. There is a glossary at the back to help you sort out all of the acronyms and to clarify terminology.

More detailed information about the Java APIs for XML is available at the following URL:

```
http://java.sun.com/xml
```

What Is XML?

XML (eXtensible Markup Language) is an industry-standard, system-independent way of representing data. Like HTML (HyperText Markup Language), XML encloses data in tags, but there are significant differences between the two markup languages. First, XML tags relate to the meaning of the enclosed text, whereas HTML tags specify how to display the enclosed text. The following XML example shows a price list with the name and price of two coffees.

```
<priceList>
    <coffee>
        <name>Mocha Java</name>
        <price>11.95</price>
    </coffee>
    <coffee>
        <name>Sumatra</name>
        <price>12.50</price>
    </coffee>
</priceList>
```

The <coffee> and </coffee> tags tell a parser that the information between them is about a coffee. The two other tags inside the <coffee> tags specify that the enclosed information is the coffee's name and its price per pound. Because XML tags indicate the content and structure of the data they enclose, they make it possible to do things like archiving and searching.

A second major difference between XML and HTML is that XML tags are extensible, allowing you to write your own XML tags to describe your content. With HTML, you are limited to using only those tags that have been predefined in the HTML specification.

With the extensibility that XML provides, you can create the tags you need for a particular type of document. You define the tags using an XML schema language. A schema describes the structure of a set of XML documents and can be used to constrain the contents of the XML documents. Probably the most-widely used schema language is the Document Type Definition schema language. A schema written in this language is called a DTD. The DTD that follows defines the tags used in the price list XML document. It specifies four tags (elements) and further specifies which tags may occur (or are required to occur) in other tags. The DTD also defines the hierarchical structure of an XML document, including the order in which the tags must occur.

```
<!ELEMENT priceList (coffee)+>
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

The first line in the example gives the highest level element, priceList, which means that all the other tags in the document will come between the <priceList> and </priceList> tags. The first line also says that the priceList element must contain one or more coffee elements (indicated by the plus sign). The second line specifies that each coffee element must contain both a name element and a price element, in that order. The third and fourth lines specify that the data between the tags <name> and </name> and between <price> and </price> is character data that should be parsed. The name and price of each coffee are the actual text that makes up the price list.

What Makes XML Portable?

A DTD, such as the priceList DTD, is what gives XML data its portability. If an application is sent a priceList document in XML format and has the priceList DTD, it can process the document according to the rules specified in the DTD. For example, given the priceList DTD, a parser will know the structure and type of content for any XML document based on that DTD. If the parser is a validating parser, it will know that the document is not valid if it contains an element not included in the DTD, such as <tea>, or if the price element precedes the name element.

Other features also contribute to the popularity of XML as a method for data interchange. For one thing, it is written in a text format, which is readable by both human beings and text-editing software. Applications can parse and process XML documents, and human beings can also read them in case there is an error in processing. Another feature is that because an XML document does not include formatting instructions, it can be displayed in various ways. Keeping data separate from formatting instructions means that the same data can be published to different media.

Overview of the Java APIs for XML

The Java APIs for XML let you write your web applications entirely in the Java programming language. They fall into two broad categories: those that deal directly with XML documents and those that deal with procedures.

- Document-oriented
    - Java™ API for XML Processing (JAXP) — processes XML documents using various parsers
    - Java™ Architecture for XML Binding (JAXB) — maps XML elements to classes in the Java programming language
- Procedure-oriented
    - Java™ API for XML Messaging (JAXM) — sends SOAP messages over the Internet in a standard way
    - Java™ API for XML Registries (JAXR) — provides a standard way to access business registries and share information
    - Java™ API for XML-based RPC (JAX-RPC) — sends SOAP method calls to remote parties over the Internet and receives the results

Perhaps the most important feature of the Java APIs for XML is that they all support industry standards, thus ensuring interoperability. Various network interoperability standards groups, such as the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS), have been defining standard ways of doing things so that businesses who follow these standards can make their data and applications work together.

Another feature of the Java APIs for XML is that they they allow a great deal of flexibility. Users have flexibility in how they use the APIs. For example, JAXP code can use various tools for processing an XML document, and JAXM code can use various messaging protocols on top of SOAP. Implementers have flexibility as well. The Java APIs for XML define strict compatibility requirements to ensure that all implementations deliver the standard functionality, but they also give developers a great deal of freedom to provide implementations tailored to specific uses.

The following sections discuss each of these APIs, giving an overview and a feel for how to use them.

JAXP

Overview

The Java™ API for XML Processing (JAXP) makes it easy to process XML data with applications written in the Java programming language. JAXP leverages the parser standards SAX (Simple API for XML Parsing) and DOM (Document Object Model) so that you can choose to parse your data as a stream of events or to build an object representation of it. JAXP version 1.1 also supports the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with DTDs that might otherwise have naming conflicts.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a pluggability layer, which allows you to plug in an implementation of the SAX or DOM APIs. The pluggability layer also allows you to plug in an XSL processor, letting you control how your XML data is displayed. The JAXP 1.1 Reference Implementation (available from `http://java.sun.com/xml`) provides the Xalan XSLT processor and the Crimson parser, both developed jointly between Sun and the Apache Software Foundation, which provides open source software.

The SAX API

SAX defines an API for an event-based parser. Being event-based means that the parser reads an XML document from beginning to end, and each time it recognizes a syntax construction, it notifies the application that is running it. The SAX parser notifies the application by calling methods from the `ContentHandler` interface. For example, when the parser comes to a less than symbol ("<"), it calls the `startElement` method; when it comes to character data, it calls the `characters` method; when it comes to the less than symbol followed by a slash ("</"), it calls the `endElement` method, and so on. To illustrate, let's look at part of the example XML document from the first section and walk through what the parser does for each line. (For simplicity, calls to the method `ignorableWhiteSpace` are not included.)

```
<priceList>     [parser calls startElement]
    <coffee>     [parser calls startElement]
        <name>Mocha Java</name>   [parser calls startElement, characters, and endElement]
        <price>11.95</price>    [parser calls startElement, characters, and endElement]
    </coffee>    [parser calls endElement]
```

The default implementations of the methods that the parser calls do nothing, so you need to write a subclass implementing the appropriate methods to get the functionality you want. For example, suppose you want to get the price per pound for Mocha Java. You would write a class extending `DefaultHandler` (the default implementation of `ContentHandler`) in which you write your own implementations of the methods `startElement` and `characters`.

You first need to create a `SAXParser` object from a `SAXParserFactory` object. You would call the method `parse` on it, passing it the price list and an instance of your new handler class (with its new implementations of the methods `startElement` and `characters`). In this example, the price list is a file, but the `parse` method can also take a variety of other input sources, including an `InputStream` object, a URL, and an `InputSource` object.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse("priceList.xml", handler);
```

The result of calling the method `parse` depends, of course, on how the methods in *`handler`* were implemented. The SAX parser will go through the file `priceList.xml` line by line, calling the appropriate methods. In addition to the methods already mentioned, the parser will call other methods such as `startDocument`, `endDocument`, `ignorableWhiteSpace`, and `processingInstructions`, but these methods still have their default implementations and thus do nothing.

The following method definitions show one way to implement the methods `characters` and `startElement` so that they find the price for Mocha Java and print it out. Because of the way the SAX parser works, these two methods work together to look for the `name` element, the characters "Mocha Java", and the `price` element immediately following Mocha Java. These methods use three flags to keep track of which conditions have been met. Note that the SAX parser will have to invoke both methods more than once before the conditions for printing the price are met.

```
public void startElement(..., String elementName, ...){
    if(elementName.equals("name")){
         inName = true;
    } else if(elementName.equals("price") && inMochaJava ){
        inPrice = true;
        inName = false;
    }
}

public void characters(char [] buf, int offset, int len) {
    String s = new String(buf, offset, len);
    if (inName && s.equals("Mocha Java")) {
        inMochaJava = true;
        inName = false;
    } else if (inPrice) {
        System.out.println("The price of Mocha Java is: " + s);
        inMochaJava = false;
        inPrice = false;
        }
    }
}
```

Once the parser has come to the Mocha Java coffee element, here is the relevant state after the following method calls:

next invocation of `startElement` -- `inName` is `true`
next invocation of `characters` -- `inMochaJava` is `true`
next invocation of `startElement` -- `inPrice` is `true`
next invocation of `characters` -- prints price

The SAX parser can perform validation while it is parsing XML data, which means that it checks that the data follows the rules specified in the XML document's DTD. A SAX parser will be validating if it is

created by a SAXParserFactory object that has had validation turned on. This is done for the SAXParserFactory object factory in the following line of code .

```
factory.setValidating(true);
```

So that the parser knows which DTD to use for validation, the XML document must refer to the DTD in its DOCTYPE declaration. The DOCTYPE declaration should be similar to this:

```
<!DOCTYPE PriceList SYSTEM "priceList.DTD">
```

The DOM API

The Document Object Model (DOM) API, defined by the W3C DOM Working Group, is a set of interfaces for building an object representation, in the form of a tree, of a parsed XML document. Once you build the DOM, you can manipulate it with DOM methods such as insert and remove, just as you would manipulate any other tree data structure. Thus, unlike a SAX parser, a DOM parser allows random access to particular pieces of data in an XML document. Another difference is that with a SAX parser, you can only read an XML document, but with a DOM parser, you can build an object representation of the document and manipulate it in memory, adding a new element or deleting an existing one.

In the previous example, we used a SAX parser to look for just one piece of data in a document. Using a DOM parser would have required having the whole document object model in memory, which is generally less efficient for searches involving just a few items, especially if the document is large. In the next example, we add a new coffee to the price list using a DOM parser. We cannot use a SAX parser for modifying the price list because it only reads data.

Let's suppose that you want to add Kona coffee to the price list. You would read the XML price list file into a DOM and then insert the new coffee element, with its name and price. The following code fragment creates a DocumentBuilderFactory object, which is then useds to create the DocumentBuilder object *builder*. The code then calls the parse method on *builder*, passing it the file priceList.xml.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse("priceList.xml");
```

At this point, *document* is a DOM representation of the price list sitting in memory. The following code fragment adds a new coffee (with the name "Kona" and the price 13.50) to the price list document. Because we want to add the new coffee right before the coffee whose name is "Mocha Java", the first step is to get a list of the name elements and iterate through the list to find "Mocha Java". Using the Node interface included in the org.w3c.dom package, the code then creates a Node object for the new coffee element and also new nodes for the name and price elements. The name and price elements contain character data, so the code creates a TextNode object for each of them and appends the text nodes to the nodes representing the name and price elements.

```
NodeList list = document.getElementsByTagName("name");
Node thisNode = list.getItem("name");
// loop through list
Node thisChild = thisNode.getChildNode();
if(thisNode.getFirstChild() instanceof org.w3c.dom.TextNode) {
```

```
        String data = thisNode.getFirstChild().getData();
    }
    if (data.equals("Mocha Java")) { // new node will be inserted before Mocha Java
        Node newNode = document.createElement("coffee");
        Node nameNode = document.createElement("name");
        TextNode textNode = document.createTextNode("Kona");
        nameNode.appendChild(textNode);

        Node priceNode = document.createElement("price");
        TextNode tpNode = document.createTextNode("13.50");
        priceNode.appendChild(tpNode);

        newNode.appendChild(nameNode);
        newNode.appendChild(priceNode);
        thisNode.insertBefore(newNode, thisNode);
    }
```

You get a DOM parser that is validating the same way you get a SAX parser that is validating: You call `setValidating(true)` on a DOM parser factory before using it to create your DOM parser, and you make sure that the XML document being parsed refers to its DTD in the DOCTYPE declaration.

XML Namespaces

All the names in a DTD are unique, thus avoiding ambiguity. However, if a particular XML document references more than one DTD, there is a possibility that two or more DTDs contain the same name. Therefore, the document needs to specify a namespace for each DTD so that the parser knows which definition to use when it is parsing an instance of a particular DTD.

There is a standard notation for declaring an XML Namespace, which is usually done in the root element of an XML document. In the following example namespace declaration, the notation `xmlns` identifies `nsName` as a namespace, and `nsName` is set to the URL of the actual namespace:

```
<priceList xmlns:nsName="myDTD.dtd"
           xmlns:otherNsName="myOtherDTD.dtd">
...
</priceList>
```

Within the document, you can specify which namespace an element belongs to as follows:

```
<nsName:price> ...
```

To make your SAX or DOM parser able to recognize namespaces, you call the method `setNamespaceAware(true)` on your `ParserFactory` instance. After this method call, any parser that the parser factory creates will be namespace aware.

The XSLT API

XSLT (XSL Transformations), defined by the W3C XSL Working Group, describes a language for transforming XML documents into other XML documents or into other formats. To perform the transfor-

mation, you usually need to supply a stylesheet, which is written in the XML Stylesheet Language (XSL). The XSL stylesheet specifies how the XML data will be displayed. XSLT uses the formatting instructions in the stylesheet to perform the transformation. The converted document can be another XML document or a document in another format, such as HTML.

JAXP supports XSLT with the `javax.xml.transform` package, which allows you to plug in an XSLT transformer to perform transformations. The subpackages have SAX-, DOM-, and stream-specific APIs that allow you to perform transformations directly from DOM trees and SAX events. The following two examples illustrate how to create an XML document from a DOM tree and how to transform the resulting XML document into HTML using an XSL stylesheet.

Transforming a DOM Tree to an XML Document

To transform the DOM tree created in the previous section to an XML document, the following code fragment first creates a `Transformer` object that will perform the transformation.

```
TransformerFactory transFactory = TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
```

Using the DOM tree root node, the following line of code constructs a `DOMSource` object as the source of the transformation.

```
DOMSource source = new DOMSource(document);
```

The following code fragment creates a `StreamResult` object to take the results of the transformation and transforms the tree to XML.

```
File newXML = new File("newXML.xml");
FileOutputStream os = new FileOutputStream(newXML);
StreamResult result = new StreamResult(os);
transformer.transform(source, result);
```

Transforming an XML Document to an HTML Document

You can also use XSLT to convert the new XML document, `newXML.xml`, to HTML using a stylesheet. When writing a stylesheet, you use XML Namespaces to reference the XSL constructs. For example, each stylesheet has a root element identifying the stylesheet language, as shown in the following line of code.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

When referring to a particular construct in the stylesheet language, you use the namespace prefix followed by a colon and the particular construct to apply. For example, the following piece of stylesheet indicates that the name data must be inserted into a row of an HTML table.

```
<xsl:template match="name">
        <tr><td>
            <xsl:apply-templates/>
        </td></tr>
</xsl:template>
```

The following stylesheet specifies that the XML data is converted to HTML and that the coffee entries are inserted into a row in a table.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="priceList">
        <html><head>Coffee Prices</head>
            <body>
                <table>
                    <xsl:apply-templates />
                </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="name">
        <tr><td>
            <xsl:apply-templates />
        </td></tr>
    </xsl:template>
    <xsl:template match="price">
        <tr><td>
            <xsl:apply-templates />
        </td></tr>
    </xsl:template>
</xsl:stylesheet>
```

To perform the transformation, you need to obtain an XSLT transformer and use it to apply the stylesheet to the XML data. The following code fragment obtains a transformer by instantiating a `TransformerFactory` object, reading in the stylesheet and XML files, creating a file for the HTML output, and then finally obtaining the `Transformer` object *`transformer`* from the `TransformerFactory` object *`tFactory`*.

```
TransformerFactory tFactory = TransformerFactory.newInstance();
String stylesheet = "prices.xsl";
String sourceId = "newXML.xml";
File pricesHTML = new File("pricesHTML.html");
FileOutputStream os = new FileOutputStream(pricesHTML);
Transformer transformer = tFactory.newTransformer(new StreamSource(stylesheet));
```

The transformation is accomplished by invoking the `transform` method, passing it the the data and the output stream.

```
transformer.transform(new StreamSource(sourceId), new StreamResult(os));
```

JAXB

JAXB provides a fast, convenient way to create a two-way mapping between XML documents and Java objects. Given a DTD, the JAXB compiler generates a set of Java classes, which contain all of the code to parse XML documents based on the schema. A developer using the generated classes can build a Java object tree representing an XML document, manipulate the content of the tree, and re-generate XML documents from the tree.

To start using a JAXB application all you need is a schema, which for the current version of JAXB, must be a DTD. You can write your own DTD, or you can obtain it from somewhere else, such as a standard DTD repository accessed through JAXR.

Once you have your DTD, you bind it to a set of classes by performing these steps:

1. Writing a binding schema, which contains the instructions on how to bind the schema to classes. The binding schema is written in a XML-based binding language, which is included with JAXB.
2. Running the schema compiler, which takes a DTD and a binding schema and generates classes from them. Each class that the schema compiler generates has `get` and `set` methods. When an instance of the class is created and initialized with data, you can use these accessor methods to access the data. A set of accessor methods is called a property.

Generating Classes From a DTD

As an example of generating classes from a DTD, consider the following DTD, which is called `priceList.dtd`.

```
<!ELEMENT priceList (coffee)+ >
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

The JAXB schema compiler is powerful enough to make reasonable assumptions from the DTD and a binding schema that specifies only the root element of the document. All you need to specify in the binding schema is that the `price` element is converted to a property that returns and accepts a `BigDecimal`:

```
...
<element name="priceList" type="class" class="PriceList" root="true" />
<element name="price" type="value" convert="BigDecimal" />
<conversion name="BigDecimal" type="java.math.BigDecimal" />
...
```

From this DTD and binding schema, the schema compiler generates a `PriceList` class and a `Coffee` class.

The `PriceList` class includes a constructor and a `List` property, to which the `coffee` element is bound.

The `Coffee` class contains a constructor and a property to represent the name of the coffee and a property to represent the price. The price accessor methods are:

```
BigDecimal getPrice();
void setPrice(BigDecimal x);
```

Both `PriceList` and `Coffee` also contain methods for unmarshalling, validating, and marshalling. Unmarshalling is the process of building an object representation of XML data. Validation is the process of checking whether the objects conform to the DTD's specifications. Marshalling is the process of generating XML data from an object representation.

Building Object Representations of XML Data

After generating your classes, you can write a Java application using the classes and build object representations of XML documents that are valid with respect to the DTD. Each object corresponds to an element in the XML document. Similarly, each object is an instance of a class from the set of generated classes. Because the objects map to both the document and the classes, you have two different ways to build the Java object tree: by unmarshalling a valid XML document or by instantiating objects from the classes. In this way, JAXB allows you to both process existing XML documents and create new XML data by instantiating the generated classes.

Suppose you have this XML document:

```
<priceList>
    <coffee>
        <name>Arabica</name>
        <price>13.50</price>
    </coffee>
    <coffee>
        <name>Mocha Java</name>
        <price>11.95</price>
    </coffee>
    <coffee>
        <name>Sumatra</name>
        <price>12.50</price>
    </coffee>
</priceList>
```

To unmarshal this XML document, you create an input stream from it and invoke the `unmarshal` method of the `PriceList` class:

```
FileInputStream fis = new FileInputStream("priceList.xml");
PriceList myPrices = PriceList.unmarshal(fis);
```

You now have a Java object tree with the `myPrices` object as the root of the tree.

Suppose you want to create your own list of coffee prices as an XML document. You first create the object tree by instantiation and then marshal the tree to an XML document. To create an object tree using instantiation, create a new `PriceList` object, get the list of `Coffee` objects from it, create a new `Coffee` object, and add it to the list:

```
PriceList myNewPrices = new PriceList();
List listOfCoffees = myNewPrices.getCoffees();
Coffee zCoffee = new Coffee();
zCoffee.setName("Zapoteca");
zCoffee.setPrice("15.00");
listOfCoffees.add(zCoffee);
```

Once you have the XML data in the form of an object tree, you can work with the data just as you would with any Java object. In this way, JAXB provides both a Java programming interface of XML data and allows seamless integration of XML data into Java applications.

Accessing Data From the Object Tree

Suppose that you want to change the price of Mocha Java in the first object tree you created. All you do is find Mocha Java in the list of coffees and set the new price by invoking `setPrice` on the `Coffee` object:

```
List coffees = myPrices.getCoffees();
for (ListIterator i = coffees.listIterator(); i.hasNext(); ) {
       Coffee myCoffee = (Coffee) i.next();
       if (myCoffee.getName().equals("Mocha Java") ) {
           myCoffee.setPrice("12.50");
       }
}
```

Writing XML Documents From the Object Tree

Whether you used unmarshalling or instantiation to build your object representation, you can marshal the objects out to an XML document, which means that JAXB also allows you to create brand new XML documents that are valid with respect to the schema.

To marshal your modified object tree to a new XML document, create an XML file and an output stream for it and invoke the `unmarshal` method on the `myNewPrices` object:

```
File newPrices = new File("newPriceList.xml");
FileOutputStream fos = new FileOutputStream(newPrices);
myNewPrices.marshal(fos);
```

Summary

JAXB essentially provides a bridge between XML and Java technology. Just as an XML document is an instance of a schema, a Java object is an instance of a class. Thus, JAXB allows you to create Java objects at the same conceptual level as the XML data. Representing your data in this way allows you to manipulate it in the same manner as you would manipulate Java objects, making it easier to create applications to process XML data. Once you have your data in the form of Java objects, it is easy to access it. In addition, after working with the data, you can write the Java objects to a new, valid XML document. With the easy access to XML data that JAXB provides, you are free to write the applications that will actually use the data, rather than spend time writing code to parse and process the data.

Differences Between JAXP and JAXB

JAXP and JAXB serve very different purposes. Which architecture or API you choose depends on the requirements of your application. One advantage of JAXP is that it allows you to parse and process your data from the same set of APIs. If you only want to grab a piece of data from a large document, you should use a SAX parser because it parses data as a stream, which is very fast. If you have a document that is not too large, and you want to add or remove data from it, you should use DOM. Although a DOM tree can occupy a large amount of memory, the DOM API includes common tree-manipulation functions. If you want to transform the data to another format, you should use JAXP, which includes the transformer API and an XSLT transform in the reference implementation that allows you to transform XML documents, SAX events, or DOM trees. JAXP also allows you the flexibility of choosing to validate the data or not.

If you want to build an object representation of XML data, but you need to get around the memory limitations of DOM, you should use JAXB. Classes created with JAXB do not include tree-manipulation capability, which is one factor contributing to the small memory footprint of a JAXB object tree. Another advantage of this kind of object tree is that you can append trees together such that a child object can have more than one parent object. In addition, processing data with JAXB is about as fast as processing it with a SAX parser because the generated classes contain all of the DTD logic, thereby avoiding the dynamic interpretation that a SAX parser must perform. The fact that JAXB requires a DTD does make it less flexible than JAXP, but this requirement guarantees that only valid data is processed. This guarantee is very important, especially if an application is receiving the data from another source. If an application doesn't have the DTD, it can't determine the meaning of the data and how it should be processed. Another advantage JAXB has over JAXP is that it allows you to specify how your code is generated from your DTD, including the data types that an element binding will accept and return.

Clearly, JAXP and JAXB have their own advantages and disadvantages, which you need to consider when choosing an XML data processing API or architecture. These two bullet lists summarize the advantages of JAXB and JAXP so that you can decide which one is right for you.

Use JAXB when you want to:

- Access data in memory, but do not need tree manipulation capabilities
- Process only data that is valid
- Convert data to different types
- Generate classes based on a DTD
- Build object representations of XML data.

Use JAXP when you want to:

- Have flexibility with regard to the way you access the data: either serially with SAX or randomly in memory with DOM
- Use your same processing code with documents based on different DTDs
- Parse documents that are not necessarily valid
- Apply XSLT transforms
- Insert or remove objects from an object tree that represents XML data

JAXM

Overview

The Java™ API for XML Messaging (JAXM) provides a standard way to send messages over the Internet from the Java platform. It is based on the SOAP 1.1 and SOAP with Attachments specifications and can be extended to work with higher level messaging protocols such as ebXML or BizTalk.

In order to do JAXM messaging, a business uses a messaging provider service, which does the behind-the-scenes work required to transport and route messages. The messaging provider implements the JAXM API, similar to the way a driver for a database implements the JDBC API. All JAXM messages go through the messaging provider, so when a business sends a message, the message first goes to the sender's messaging provider, then to the recipient's messaging provider, and finally to the intended recipient. It is also possible to route a message to go to intermediate recipients, called *actors*, before it goes to the ultimate destination.

Because all messages go through it, a messaging provider can take care of housekeeping details like assigning message identifiers and keeping track of whether a message has been delivered before. A messaging provider can also try resending a message that did not reach its destination on the first attempt at delivery. The beauty of a messaging provider is that the client using JAXM technology ("JAXM client") is totally unaware of what the provider is doing in the background. The JAXM client simply makes JAXM method calls, and the messaging provider, working with the container, if there is one, makes everything happen.

Though it is not required, JAXM messaging usually takes place within a container, generally a servlet or a J2EE™ container. One of the advantages of being in a container is that you can have a listener, which makes it possible to receive messages asynchronously. The listener receives the message as one operation, and the recipient sends a reply as a subsequent operation, thus making the messaging asynchronous. When there is no listener, messages are sent via a method that blocks until it gets a reply. This kind of messaging is synchronous, meaning that sending a message and receiving a reply are all one continuous operation and that nothing else can happen until the operation is completed.

A JAXM message is made up of two parts, a required SOAP part and an optional attachment part. The SOAP part, which consists of a `SOAPEnvelope` object containing a `SOAPHeader` object and a `SOAPBody` object, can hold an XML document as the content of the message being sent. If you want to send multiple documents or content that is not an XML document, your message will need to contain an attachment part. There is no limitation on the content in the attachment part, so you can send images or any other kind of content.

Creating a Message

Getting a Connection to the Messaging Provider

The first thing a JAXM client must do is get a connection to its messaging provider. It uses the connection to create a `MessageFactory` object, which can then be used to create `Message` objects. Once the message has been populated, the connection will be used again to send the message.

The following code demonstrates getting a message factory and using it to create a connection. The first two lines use the JNDI API to retrieve the appropriate `ConnectionFactory` object from the naming service where it was registered with the name "CoffeeBreakProvider". When this logical name is passed as an argument, the method `lookup` returns the `ConnectionFactory` object to which the logical name was bound.

The value returned is a Java `Object`, which must be narrowed to a `ConnectionFactory` object so that it can be used to create a connection. The third line uses a JAXM method to actually get the connection.

```
Context ctx = getInitialContext();
ConnectionFactory cf = (ConnectionFactory)ctx.lookup("CoffeeBreakProvider");
Connection con = cf.getConnection();
```

The `Connection` instance *con* represents a connection to The Coffee Break's messaging provider. In the following lines of code, it is used to create a `MessageFactory` object, which is then used to create a `Message` object.

```
MessageFactory messageFactory = con.getMessageFactory();
Message m = messageFactory.createMessage();
```

Part of the flexibility of the JAXM API is that it allows a specific usage of a SOAP header. For example, ebXML or BizTalk protocols can be built on top of SOAP messaging. This usage of SOAP by a given standards group or industry is called a *profile*. If the method `getMessageFactory` is given no argument, as was done in the preceding code fragment, it will return a `MessageFactory` object that produces messages that use the base SOAP profile. Thus, the `Message` object *m* created in the preceding line of code will support a basic SOAP message. If a profile is passed to the method `getMessageFactory`, the `Message` objects created by the resulting `MessageFactory` object will support the specified profile. For example, in the following code fragment, *m2* will support the messaging profile that is supplied to `getMessageFactory`.

```
MessageFactory messageFactory2 = con.getMessageFactory(<profile>);
Message m2 = messageFactory2.createMessage();
```

Each of the new `Message` objects *m* and *m2* automatically contains the required `SOAPPart` object, but its header and body do not yet have any content. The following sections will illustrate typical ways to add content.

Populating a Message

There are two ways to add content to a message:

1. Passing a `javax.xml.transform.Source` object to a `SOAPEnvelope` object. The `Source` object can be a `SAXSource` object, a `DOMSource` object, or a `StreamSource` object. The `Source` object contains the content for the message and also the information needed for it to act as source input. A `StreamSource` object will contain the content as an XML document; the `SAXSource` or `DOMSource` object will contain content and instructions for transforming it into an XML document.

2. Creating separate elements containing the content and adding them individually. In this case, you build an XML document using `String`, `COMMENT`, and `CDATA` objects as needed.

Populating the SOAP Part of a Message

As stated earlier, all messages are created with a `SOAPPart` object, which has a `SOAPEnvelope` object containing a `SOAPHeader` object and a `SOAPBody` object. One way to add content to the SOAP part of a message is to create a `SOAPHeaderElement` object or a `SOAPBodyElement` object and add an XML document that

you build with String, CDATA, and COMMENT objects. Another way is to add content to the SOAPEnvelope object by passing it a javax.xml.transform.Source object, which may be a SAXSource, DOMSource, or StreamSource object.

The following code fragment illustrates adding content as a DOMSource object. The first step is always to get the SOAPPart object from the Message object and use it to get the SOAPEnvelope object.

```
SOAPPart soapPart = m.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getSOAPEnvelope();
```

Next the code builds the XML document to be added. It uses a DocumentBuilderFactory object to get a DocumentBuilder object. Then it parses the given file to produce the document that will be used to initialize a new DOMSource object. Finally, the code passes the DOMSource object *domSource* to the SOAPEnvelope object.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("file:///foo.bar/soap.xml");
DOMSource domSource = new DOMSource(doc);

soapEnvelope.setContent(domsrc);
```

Populating the Attachment Part of a Message

A Message object may have no attachment parts, but if it has anything that is not an XML document, it must have an attachment part. There may be any number of attachment parts, and they may contain anything from plain text (including XML documents) to image files. In the following code fragment, the content is an image in a JPEG file, whose URL is used to initialize the javax.activation.DataHandler object *dh*. The Message object *m* creates the AttachmentPart object *attachPart*, which is initialized with the data handler containing the URL for the image. Finally, the message adds attachPart to itself.

```
URL url = new URL("http://foo.bar/img.jpg");
DataHandler dh = new DataHandler(url);
AttachmentPart attachPart = m.createAttachmentPart(dh);
m.addAttachmentPart(attachPart);
```

Sending a Message

Once you have populated a Message object, you are ready to send it by using a Connection object (a connection to your messaging provider). If messaging is asynchronous, that is, being done in the context of a container such as a servlet or a J2EE container, the method send is used. If the application sending the message is not running in a container, which means that messaging is synchronous, the method call is used. The parameters to both methods are (1) the Message object that is being sent and (2) the Endpoint object representing the destination to which it is being sent.

The following code fragment sends the message asynchronously because it uses the method send. It creates an Endpoint object *endPoint* from a URI for the intended recipient and then passes *endPoint* and the Message object *m* to the method send.

```
Endpoint endPoint = new Endpoint("http://foo.bar/Service");
con.send(m, endPoint);
```

If your application is not running in a container, you need to use the method `call` to send a message. This method takes the same parameters as the method `send`, but, unlike `send`, it will block until it receives a reply and returns it, as shown in the following line of code.

```
Message reply = con.call(m, endPoint);
```

JAXR

Overview

The Java™ API for XML Registries (JAXR) provides a convenient way to access standard business registries over the Internet. Business registries are often described as electronic yellow pages because they contain listings of businesses and the products or services the businesses offer. JAXR gives developers writing applications in the Java programming language a uniform way to use business registries that are based on open standards (such as ebXML) or industry consortium-led specifications (such as UDDI).

Businesses can register themselves with a registry or discover other businesses with whom they might want to do business. In addition, they can submit material to be shared and search for material that others have submitted. Standards groups have developed DTDs for particular kinds of XML documents, and two businesses might, for example, agree to use the DTD for their industry's standard purchase order form. Because the DTD is stored in a standard business registry, both parties can use JAXR to access it.

Registries are becoming an increasingly important component of web services because they allow businesses to collaborate with each other dynamically in a loosely coupled way. Accordingly, the need for JAXR, which enables enterprises to access standard business registries from the Java programming language, is also growing.

Using JAXR

The following sections give examples of two of the typical ways a business registry is used. They are meant to give you an idea of how to use JAXR rather than to be complete or exhaustive.

Registering a Business

An organization that uses the Java platform for its electronic business would use JAXR to register itself in a standard registry. It would supply its name, a description of itself, and some classification concepts to facilitate searching for it. This is shown in the following code fragment, which first creates the `RegistryService` object *`rs`* and then uses it to create the `BusinessLifeCycleManager` object *`lcm`*. The business, a chain of coffee houses called The Coffee Break, is represented by the `Organization` object *`org`*, to which The Coffee Break adds its name, a description of itself, and some categories for its classification. Then *`org`*, which now contains the properties and classification concepts for The Coffee Break is added to the `Collection` object *`orgs`*. Finally, *`orgs`* is saved by *`lcm`*, which will manage the life cycle of the `Organization` objects contained in *`orgs`*.

```
RegistryService rs = connection.getRegistryService();

BusinessLifeCycleManager lcm = rs.getBusinessLifeCycleManager();
Collection orgs = new ArrayList();
while (...) {
    Organization org = new Organization();
    org.setName("The Coffee Break");
    org.setDescription("Purveyer of only the finest coffees. Established 1895");
    .....
    Collection classificationConcepts = new ArrayList();
    classificationConcepts.add(new Key(<key for Industry/Food Manufacturing/
                    Other Food Manufacturing/Coffee And Tea Manufacturing concept>));
```

```
    classificationConcepts.add(new Key(<key for Geography/North America concept>));
    ....
    org.addClassifications(classificationConcepts);
    orgs.add(org);
}
lcm.saveOrganizations(orgs);
```

Searching a Registry

A business can also use JAXR to search a registry for other businesses. The following code fragment uses the `BusinessQueryManager` object *bqm* to search The Coffee Break. Before *bqm* can invoke the method `findOrganizations`, the code needs to define the search criteria to be used. In this case, three of the possible six search parameters are supplied to `findOrganizations`; because `null` is supplied for the third, fifth, and sixth parameters, those criteria are not used to limit the search. The first, second, and fourth arguments are all `Collection` objects, with *findQualifiers* and *names* being defined here. The only element in *findQualifiers* is a `String` specifying that no organization be returned unless its name is an exact match to one of the names in the *names* parameter. This parameter, which is also a `Collection` object with only one element, says that businesses with "Coffee" in their names are a match. The other `Collection` object is *classificationConcepts*, which was defined when The Coffee Break registered itself. The previous code fragment, in which the industry and geographical location for The Coffee Break were provided, is an example of defining classification concepts.

```
BusinessQueryManager bqm = rs.getBusinessQueryManager();

//Define find qualifiers
Collection findQualifiers = new ArrayList();
findQualifiers.add(BusinesQueryManager.exactNameMatch);
Collection names = new ArrayList();
names.add("%Coffee%"); //Find orgs with name containing 'Coffee'

//Find using only the name and the classification concepts
BulkResponse response = bqm.findOrganizations(findQualifiers, names, null,
                                              classificationConcepts, null, null);
Collection orgs = response.getCollection();
```

JAXR also supports using an SQL query to search a registry. This is done using an `SQLQueryManager` object, as the following code fragment demonstrates.

```
SQLQueryManager sqm = rs.getSQLQueryManager();
BulkResponse response2 = sqm.submitQuery("SELECT id FROM RegistryEntry WHERE
                                         name LIKE %Coffee% AND majorVersion >= 1 AND
                                         (majorVersion >= 2 OR minorVersion >= 3)");
```

The `BulkResponse` object *response2* will contain a value for `id` (a uuid) for each entry in `RegistryEntry` that has Coffee in its name and also has a version number of 1.3 or greater.

To ensure interoperable communication between a JAXR client and a registry implementation, the messaging is done using JAXM. This is done completely behind the scenes, so as a user of JAXR, you are not even aware of it.

Because JAXM supports all the major registry standards, you can use it to access a variety of registries, including ebXML and UDDI registries.

JAX-RPC

Overview

The Java™ API for XML-based RPC (JAX-RPC) makes it possible to write an application in the Java programming language that uses XML to make a remote procedure call (RPC).

The Java programming language already has two other APIs for making remote procedure calls, Java IDL and Remote Method Invocation (RMI). All three have an API for marshalling and unmarshalling arguments and for transmitting and receiving procedure calls. The difference is that JAX-RPC is based on XML and is geared to web services. Java IDL is based on the Common Object Request Broker Architecture (CORBA) and uses the Object Management Group's Interface Definition Language (OMG IDL). RMI is based on RPC where both the method calls and the methods being invoked are in the Java programming language--although with RMI over IIOP, the methods being invoked may be in another language. Sun will continue its support of CORBA and RMI in addition to developing JAX-RPC, as each serves a distinct need and has its own set of users.

All varieties of RPC are fairly complex, involving the mapping and reverse mapping of data types and the marshalling and unmarshalling of arguments. However, these take place behind the scenes and are not visible to the user. JAX-RPC continues this model, which means that a client using XML-based RPC from the Java programming language is not required to work with XML or do any mapping directly.

Using JAX-RPC

JAX-RPC makes using a Web service easier, and it also makes developing a Web service easier, especially if you use the J2EE platform. An RPC-based Web service is basically a collection of procedures that can be called by a remote client over the Internet. The service itself is a server application deployed on a server-side container that implements the procedures that are available for clients to call. For example, a typical RPC-based Web service is a stock quote service that takes a SOAP request for the price of a specified stock and returns the price via SOAP.

A Web service needs to make itself available to potential clients, which it can do, for instance, by describing itself using the Web Services Description Language (WSDL). A consumer (Web client) can then do a lookup of the WSDL document to access the service. A consumer using the Java programming language uses JAX-RPC to send its request to the service, which may or may not have been defined and deployed on a Java platform. The converse is also possible, that is, a client using another programming language can send its request to a service that has been defined and deployed on a Java platform.

Although JAX-RPC implements a remote procedure call as a request and a response SOAP message, a user of JAX-RPC is shielded from this level of detail. So, underneath the covers, JAX-RPC is actually a specialized form of SOAP messaging. In contrast, JAXM is a robust form of SOAP messaging, providing the developer with its full richness.

The following list includes features that JAXM can provide and that RPC, including JAX-RPC, does not generally provide:

- Asynchronous messaging
- Routing of a message to more than one party
- Reliable messaging with features such as guaranteed delivery

JAX-RPC is the better choice for applications that wish to avoid the complexities of SOAP messaging and where communication using the RPC model is a good fit. The important thing is that whether you use JAXM or JAX-RPC, you can conveniently do XML messaging using the Java programming language.

Sample Scenario

Overview

The following scenario is an example of how the Java APIs for XML might be used and how they work together. Part of the richness of the Java APIs for XML is that in many cases they offer alternate ways of doing something and thus let you tailor your code to meet individual needs. This section will point out some instances in which an alternate API could have been used and will also give the reasons why one API or the other might be a better choice.

Scenario

Suppose that the owner of a chain of coffee houses, called The Coffee Break, wants to expand the line of coffees that he sells. He instructs his business manager to find some new coffee suppliers and get their wholesale prices. The Coffee Break can analyze the prices and decide which new coffees it wants to carry and which companies it wants to buy them from. The business manager assigns the task to the company's software engineer, who decides that the best way to locate new coffee suppliers is to search the ebXML business registry, where The Coffee Break has already registered itself.

The engineer uses JAXR to send a query searching for wholesale coffee suppliers. JAXR sends messages using JAXM in the background, which ensures that the registry will be able to receive and understand it.

The ebXML registry will receive the query and apply the search criteria in the JAXR code to the information it has about the organizations registered with it. When the search is completed, the registry will send back a list of the distributors that sell wholesale coffee.

The engineer's next step is to draft a request for price lists and send it to each of the coffee distributors using JAXM. She writes an application that gets a connection to the company's messaging service so that she can send the requests. She then creates a JAXM message, adds the request, and sends it.

Each coffee distributor receives the request, and before sending out current prices, checks with its stock quote service using JAX-RPC to get the latest quotes for the relevant coffee futures. Based on the figures they get back, the distributors send The Coffee Break their newly revised prices in an XML price sheet. The vendors use XML because that way they can use a format that is convenient for them and that their buyers can process easily even if they are using many different information systems.

Compare Prices and Order Coffees

The engineer decides to use JAXB to process the price lists. The list of distributors that the registry returned contained information for getting the DTDs that the distributors use, and conveniently, they all use a standard form for the price list. Because they all use one standard DTD, the engineer can generate a set of classes from that DTD that applies to the price lists from all of the suppliers. (Otherwise, she would have used SAX or DOM to do the processing.) The engineer's application will work with each coffee as an object that has a price property and a name property. After instantiating the classes, the application gets the prices from the `Coffee` objects and compares the prices quoted by the different vendors.

When the owner and business manager decide which suppliers to do business with, based on the engineer's price comparisons, they are ready to send an order. With JAXB, the engineer creates a new XML order form based on the classes generated from the price list DTD. This new order form, which contains only the coffees that the owner wants to buy, is then sent to the suppliers via JAXM. Each supplier will acknowledge receipt of the order via JAXM.

Selling Coffees on the Internet

Meanwhile, The Coffee Break has been preparing for its expanded coffee line. It will need to publish a new price list/order form in HTML for its web site. But before that can be done, the company needs to determine what prices it will charge. The engineer uses the same objects she created for comparing prices and composing the wholesale order form to access each price and multiply it by 125% to arrive at the price that The Coffee Break will charge. With a few modifications, the list of retail prices will become the online order form.

With the objects containing the new retail prices, the engineer can use JavaServer Pages™ (JSP™) technology to create an HTML order form that customers can use to order coffee online. The engineer accesses the objects from the JSP page and inserts the name and the price of each `Coffee` object into an HTML table on the JSP page. The customer enters the quantity of each coffee he or she wants to order and clicks the Submit button to send the order.

Conclusion

Although this scenario is simplified for the sake of brevity, it illustrates how pervasive XML technologies are becoming in the world of web services. And now, with the Java APIs for XML and the J2EE platform, it keeps getting easier to implement web services and to write applications that are the consumers of web services.

Glossary

asynchronous Loosely coupled, occurring at different times. In asynchronous messaging, a message is sent, and the reply is received some time later as a separate transaction. See synchronous.

B2B Business-to-business A term used to describe web services between two businesses, such as between a wholesale supplier and a retail outlet.

B2C Business-to-customer A term used to describe web services between a business and an end user, such as between a retail outlet and a retail customer.

DOM Document Object Model A standard API for parsing XML data into an object-tree representation and manipulating the contents of the tree. DOM is being developed through the World Wide Web Consortium. JAXP provides a Java programming interface for the DOM API and allows an application to plug in a compliant DOM parser.

DTD Document Type Definition A simple type of schema that defines the kind of information in a particular type of XML document.

ebXML Electronic Business XML An open public initiative that is developing specifications aimed at enabling a single global electronic marketplace based on using XML and the Internet.

HTML HyperText Markup Language A markup language used for formatting web pages.

HTTP HyperText Transfer Protocol A protocol for transfering data over the Internet.

J2EE™ Java™ 2 Platform, Enterprise Edition The Java platform that defines the standard for multitier enterprise computing. The J2EE platform includes the J2SE platform.

J2SE™ Java™ 2 Platform, Standard Edition The Java platform for client-side computing.

JAX Pack The upcoming bundle of XML-related Java APIs (JAXP, JAXB, JAXM, JAXR, and JAX-RPC). The JAX Pack will be included in the Web Services Pack.

JAXB Java™ Architecture for XML Binding The architecture for converting data in an XML document to objects in the Java programming language. Given an XML document's schema (for example, a DTD), the JAXB compiler will produce classes that correspond to the DTD. The generated classes include methods that allow building an object tree with XML data based on the DTD and writing the tree out to a new XML document.

JAXM Java™ API for XML Messaging The standard API for sending SOAP messages using the Java programming languge. JAXM is based on SOAP 1.1 with Attachments and provides the ability to layer other profiles, such as ebXML or BizTalk, on top of it.

JAXP Java™ API for XML Processing A comprehensive API for parsing and processing XML documents. JAXP includes support for the Simple API for XML Parsing (SAX), the Document Object Model (DOM), the eXtensible Stylesheet Language for Transformations (XSLT), and XML Namespaces.

JAXR Java™ API for XML Registries The standard API for convenient access to Internet business registries from the Java platform.

JAX-RPC Java™ API for XML-based RPC The standard API for sending XML-based remote procedure calls using the Java programming language.

loosely coupled A term referring to the relationship between two businesses in which each business has no prior knowledge of, and no dependencies on, the other business's information technology infrastructure. XML is the key enabler that allows loosely coupled businesses to communicate over the Internet and thereby conduct electronic business with each other.

OASIS Organization for the Advancement of Structured Information Standards A non-profit consortium promoting the adoption of interoperable specifications of public standards, such as XML.

registry A web-based service that enables dynamic and loosely coupled business-to-business collaboration by providing access to shared information. A registry is sometimes compared to an electronic yellow pages for businesses. See repository.

repository A data storage facility much like a database. A business registry uses a repository to store its data, such as information about businesses, XML descriptions of specific business protocols (for example, RosettaNet PIP3A4

for purchase orders), and XML schemas defining the structure of XML documents exchanged during a supported business process.

schema A specification of the structure of a set of XML documents. A DTD is an example of a schema.

SAX Simple API for XML Parsing A standard API that defines an event-based XML parser. SAX was developed by members of the XML-DEV mailing list, and currently the OASIS standards body is continuing to develop the SAX API. JAXP provides a Java programming interface for the SAX API and allows an application to plug in a compliant SAX parser.

synchronous Tightly coupled, occurring at the same time. In synchronous messaging, a message is sent, and nothing else can happen until the response is sent back as part of the same process. In other words, the message and reply are tightly coupled. In the JAXM API, the method `call` is used for sending a synchronous message. It sends the message and then blocks until it gets the reply.

SOAP Simple Object Access Protocol A lightweight XML-based protocol, developed by the W3C, for the exchange of information in a decentralized, distributed environment.

UDDI Universal Description, Discovery and Integration An initiative based on standard registry services that provide Yellow, White, and Green Page business-centered functionality. It focuses on giving large organizations a way to reach out to and manage their networks of smaller business customers.

web services Content and software processes that provide services to customers over the Internet. Web services are delivered over the Internet in a loosely coupled way, using XML interfaces. For example, a service based on JAX-RPC is a collection of procedures that can be called by remote clients.

Web Services Pack A download that will bundle together key technologies to simplify building web services using the Java 2 Platform. It includes JavaServer™ Faces (a standard API for creating Java Web GUIs), Tomcat (an open-source implementation of JavaServer Pages™ and Java Servlet technologies), and JAX Pack (a bundle of the the Java APIs for XML).

WSDL Web Services Description Language A language that specifies an XML format for describing a web service.

WWW World Wide Web The web of systems and the data in them that is the Internet.

W3C World Wide Web Consortium A group of member organizations developing standard protocols for web technologies to ensure interoperability of the Web's languages and protocols.

XML Extensible Markup Language A markup language that describes the hierarchical structure of content in a document. Because XML makes data portable, it has become the standard for sharing data over the Internet between applications, distributed web services, and trading partners.

XML Namespaces A W3C standard for building documents that reference more than one DTD, more than one of which might define the same element name. JAXP provides support for XML Namespaces.

XSL eXtensible Stylesheet Language A language for specifying a stylesheet, which provides formatting instructions for XML data. To actually transform an XML document using the stylesheet, an application uses XSLT, which is an extension of XSL**.**

XSLT XSL Transformations A language for transforming XML documents to other XML documents or to documents of other formats, such as HTML. An application can use XSLT to transform documents according to the formatting instructions in an XSL stylesheet, but it can also use XSLT independently of XSL.